2

KERNEL SERVICES

The Solaris kernel manages operating system resources and provides facilities to user processes. In this chapter we explore how the kernel implements these services. We begin by discussing the boundary between user programs and kernel mode, then discuss the mechanisms used to switch between user and kernel mode, including system calls, traps, and interrupts.

2.1 Access to Kernel Services

The Solaris kernel insulates processes from kernel data structures and hardware by using two distinct processor execution modes: nonprivileged mode and privileged mode. Privileged mode is often referred to as *kernel mode;* nonprivileged mode is referred to as *user mode*.

In nonprivileged mode, a process can access only its own memory, whereas in privileged mode, access is available to all of the kernel's data structures and the underlying hardware. The kernel executes processes in nonprivileged mode to prevent user processes from accessing data structures or hardware registers that may affect other processes or the operating environment. Because only Solaris kernel instructions can execute in privileged mode, the kernel can mediate access to kernel data structures and hardware devices.

If a user process needs to access kernel system services, a thread within the process transitions from user mode to kernel mode through a set of interfaces known as *system calls*. A system call allows a thread in a user process to switch into kernel mode to perform an OS-defined system service. Figure 2.1 shows an example of a user process issuing a `read()` system call. The `read()` system call executes special machine code instructions to change the processor into privileged mode, in order to begin executing the `read()` system call's kernel instructions. While in privileged mode, the kernel `read()` code performs the I/O on behalf of the calling thread, then returns to nonprivileged user mode, after which the user thread continues normal execution.

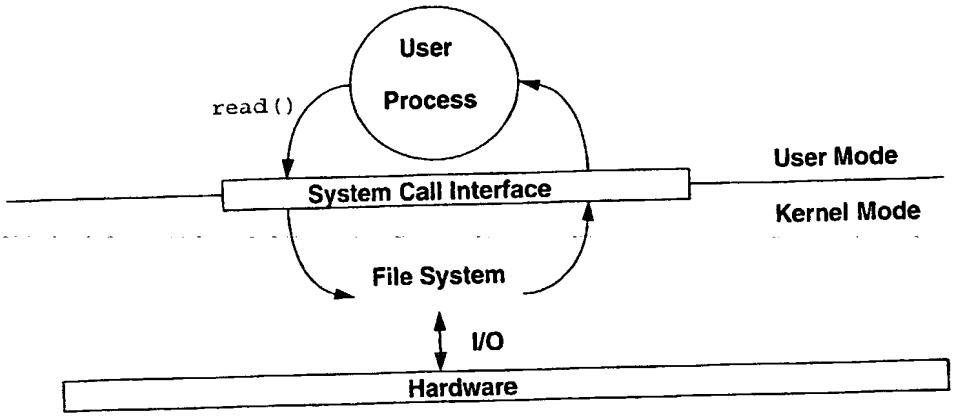

Figure 2.1 Switching into Kernel Mode via System Calls

2.2 Entering Kernel Mode

In addition to entering through system calls, the system can enter kernel mode for other reasons, such as in response to a device interrupt, or to take care of a situation that could not be handled in user mode. A transfer of control to the kernel is achieved in one of three ways:

- Through a system call
- As the result of an interrupt
- As the result of a processor trap

We defined a *system call* as the mechanism by which a user process requests a kernel service, for example, to read from a file. System calls are typically initiated from user mode by either a trap instruction or a software interrupt, depending on the microprocessor and platform. On SPARC-based platforms, system calls are initiated by issuing a specific trap instruction in a C library stub.

An *interrupt* is a vectored transfer of control into the kernel, typically initiated by a hardware device, for example, a disk controller signalling the completion of an I/O. Interrupts can also be initiated from software. Hardware interrupts typically occur asynchronously to the currently executing thread, and they occur in interrupt context.

A *trap* is also a vectored transfer of control into the kernel, initiated by the processor. The primary distinction between traps and interrupts is this: Traps typically occur as a result of the current executing thread, for example, a divide-by-zero error or a memory page fault; interrupts are asynchronous events, that is, the source of the interrupt is something unrelated to the currently executing thread. On SPARC processors, the distinction is somewhat blurred, since a trap is also the mechanism used to initiate interrupt handlers.

2.2.1 Context

A context describes the environment for a thread of execution. We often refer to two distinct types of context: an execution context (thread stacks, open file lists, resource accounting, etc.) and a virtual memory context (the virtual-to-physical address mappings).

2.2.1.1 Execution Context

Threads in the kernel can execute in process, interrupt, or kernel context.

- **Process Context** — In the process context, the kernel thread acts on behalf of the user process and has access to the process's user area (*uarea*), and process structures for resource accounting. The `uarea` (`struct u`) is a special area within the process that contains process information of interest to the kernel: typically, the process's open file list, process identification information, etc. For example, when a process executes a system call, a thread within the process transitions into kernel mode and then has access to the `uarea` of the process's data structures, so that it can pass arguments, update system time usage, etc.
- **Interrupt Context** — Interrupt threads execute in an interrupt context. They do not have access to the data structures of the process or thread they interrupted. Interrupts have their own stack and can access only kernel data structures.
- **Kernel Context** — Kernel management threads run in the kernel context. In kernel context, system management threads share the kernel's environment with each other. Kernel management threads typically cannot access process-related data. Examples of kernel management threads are the page scanner and the NFS server.

2.2.1.2 Virtual Memory Context

A virtual memory context is the set of virtual-to-physical address translations that construct a memory environment. Each process has its own virtual memory con-

text. When execution is switched from one process to another during a scheduling switch, the virtual memory context is switched to provide the new process's virtual memory environment.

On Intel and older SPARC architectures, each process context has a portion of the kernel's virtual memory mapped within it, so that a virtual memory context switch to the kernel's virtual memory context is not required when transitioning from user to kernel mode during a system call. On UltraSPARC, features of the processor and memory management unit allow fast switching between virtual memory contexts; in that way, the process and kernel can have separate virtual memory contexts. See "Virtual Address Spaces" on page 130 and "Kernel Virtual Memory Layout" on page 205 for a detailed discussion of process and kernel address spaces.

2.2.2 Threads in Kernel and Interrupt Context

In addition to providing kernel services through system-calls, the kernel must also perform system-related functions, such as responding to device I/O interrupts, performing some routine memory management, or initiating scheduler functions to switch execution from one kernel thread to another.

- **Interrupt Handlers** — Interrupts are directed to specific processors, and on reception, a processor stops executing the current thread, context-switches the thread out, and begins executing an interrupt handling routine. Kernel threads handle all but high-priority interrupts. Consequently, the kernel can minimize the amount of time spent holding critical resources, thus providing better scalability of interrupt code and lower overall interrupt response time. We discuss on kernel interrupts in more detail in "Interrupts" on page 38.
- **Kernel Management Threads** — The Solaris kernel, just like a process, has several of its own threads of execution to carry out system management tasks (the memory page scanner and NFS server are examples). Solaris kernel management threads do not execute in a process's execution context. Rather, they execute in the *kernel's* execution context, sharing the kernel execution environment with each other. Solaris kernel management threads are scheduled in the system (SYS) scheduling class at a higher priority than most other threads on the system.

Figure 2.2 shows the entry paths into the kernel for processes, interrupts, and threads.

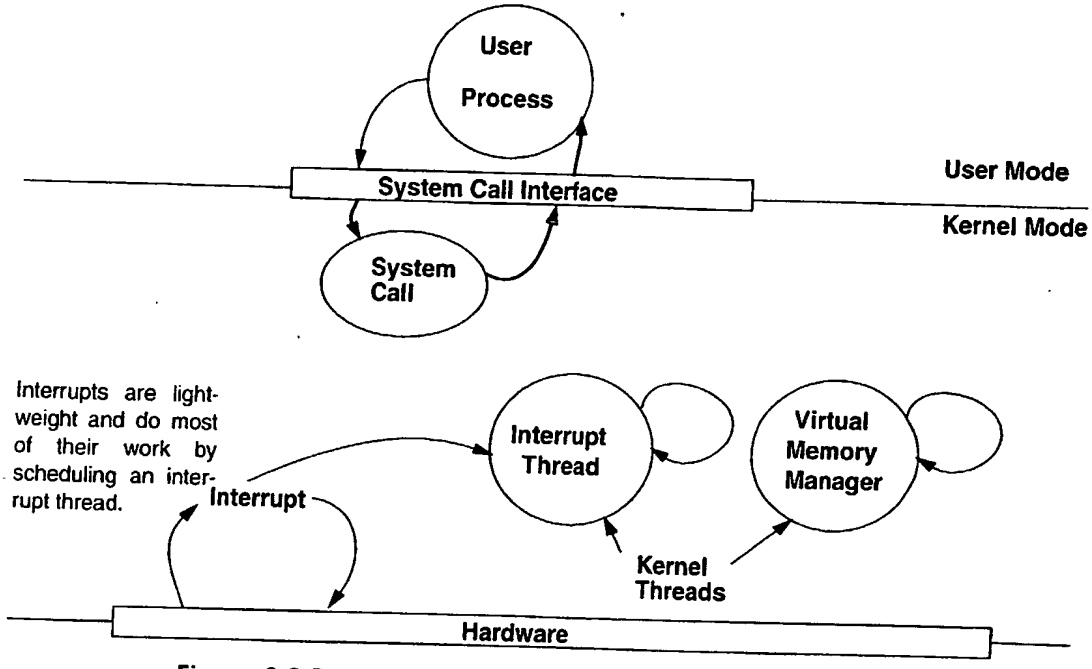

Figure 2.2 Process, Interrupt, and Kernel Threads

2.2.3 UltraSPARC I & II Traps

The SPARC processor architecture uses traps as a unified mechanism to handle system calls, processor exceptions, and interrupts. A SPARC trap is a procedure call initiated by the microprocessor as a result of a synchronous processor exception, an asynchronous processor exception, a software-initiated trap instruction, or a device interrupt.

Upon receipt of a trap, the UltraSPARC I & II processor enters privileged mode and transfers control to the instructions, starting at a predetermined location in a *trap table*. The trap handler for the type of trap received is executed, and once the interrupt handler has finished, control is returned to the interrupted thread. A trap causes the hardware to do the following:

- Save certain processor state (program counters, condition code registers, trap type, etc.)
- Enter privileged execution mode
- Begin executing code in the corresponding trap table slot

When an UltraSPARC trap handler processing is complete, it issues a SPARC DONE or RETRY instruction to return to the interrupted thread.

The UltraSPARC I & II trap table is an in-memory table that contains the first eight instructions for each type of trap. The trap table is located in memory at the address stored in the trap table base address register (TBA), which is initialized during boot. Solaris places the trap table at the base of the kernel (known as `kernelbase`) in a locked-down (non-pageable) 4-Mbyte page so that no memory-related traps (page faults or TLB misses) will occur during execution of instructions in the trap table. (For a detailed kernel memory map, see Appendix B, "Kernel Virtual Address Maps".)

2.2.3.1 UltraSPARC I & II Trap Types

The trap table contains one entry for each type of trap and provides a specific handler for each trap type. The UltraSPARC I & II traps can be categorized into the following broad types:

- **Processor resets** — Power-on reset, machine resets, software-initiated resets
- **Memory management exceptions** — MMU page faults, page protection violations, memory errors, misaligned accesses, etc.
- **Instruction exceptions** — Attempts to execute privileged instructions from nonprivileged mode, illegal instructions, etc.
- **Floating-point exceptions** — Floating-point exceptions, floating-point mode instruction attempted when floating point unit disabled, etc.
- **SPARC register management** — Traps for SPARC register window spilling, filling, or cleaning.
- **Software-initiated traps** — Traps initiated by the SPARC trap instruction (Tcc); primarily used for system call entry in Solaris.

Table 2-1 shows the UltraSPARC I & II trap types, as implemented in Solaris.

Table 2-1 Solaris UltraSPARC I & II Traps

Trap Definition	Trap Type	Priority
Power-on reset	001	0
Watchdog reset	002	1
Externally initiated reset	003	1
Software-initiated reset	004	1
RED state exception	005	1
Reserved	006...007	n/a
Instruction access exception	008	5
Instruction access MMU miss	009	2
Instruction access error	00A	3
Reserved	00B...00F	n/a
Illegal instruction	010	7

Table 2-1 Solaris UltraSPARC I & II Traps (Continued)

Trap Definition	Trap Type	Priority
Attempt to execute privileged instruction	011	6
Unimplemented load instruction	012	6
Unimplemented store instruction	013	6
Reserved	014...01F	n/a
Floating-point unit disabled	020	8
Floating-point exception ieee754	021	11
Floating-point exception – other	022	11
Tag overflow	023	14
SPARC register window clean	024...027	10
Division by zero	028	15
Internal processor error	029	4
Data access exception	030	12
Data access MMU miss	031	12
Data access error	032	12
Data access protection	033	12
Memory address not aligned	034	10
Load double memory address not aligned	035	10
Store double memory address not aligned	036	10
Privileged action	037	11
Load quad memory address not aligned	038	10
Store quad memory address not aligned	039	10
Reserved	03A...03F	n/a
Asynchronous data error	040	2
Interrupt level *n*, where *n*=1...15	041...04F	32-*n*
Reserved	050...05F	n/a
Vectored interrupts	060...07F	Int. Specific
SPARC register window overflows	080...0BF	9
SPARC register window underflows	0C0...0FF	9
Trap instructions Tcc	100...17F	16
Reserved	180...1FF	n/a

2.2.3.2 UltraSPARC I & II Trap Priority Levels

Each UltraSPARC I & II trap has an associated priority level. The processor's trap hardware uses the level to decide which trap takes precedence when more than one trap occurs on a processor at a given time. When two or more traps are pending, the highest-priority trap is taken first (0 is the highest priority).

Interrupt traps are subject to trap priority precedence. In addition, interrupt traps are compared against the *processor interrupt level* (PIL). The UltraSPARC I & II processor will only take an interrupt trap that has an *interrupt request level*

greater than that stored in the processor's PIL register. We discuss this behavior in more detail in "Interrupts" on page 38.

2.2.3.3 UltraSPARC I & II Trap Levels

The UltraSPARC I & II processor introduced nested traps; that is, a trap can be received while another trap is being handled. Prior SPARC implementations could not handle nested traps (a "watchdog reset" occurs on pre-UltraSPARC processors if a trap occurs while the processor is executing a trap handler). Also introduced was the notion of *trap levels* to describe the level of trap nesting. The nested traps have five levels, starting at trap level 0 (normal execution, no trap) through trap level 4 (trap level 4 is actually an error handling state and should not be reached during normal processing).

When an UltraSPARC I & II trap occurs, the CPU increments the trap level (TL). The most recent processor state is saved on the trap stack, and the trap handler is entered. On exit from the handler, the trap level is decremented.

UltraSPARC I & II also implements an alternate set of global registers for each trap level. Those registers remove most of the overhead associated with saving state, making it very efficient to move between trap levels.

2.2.3.4 UltraSPARC I & II Trap Table Layout

The UltraSPARC I & II trap table is halved: the lower half contains trap handlers for traps taken at trap level 0, and the upper half contains handlers for traps taken when the trap level is 1 or greater. We implement separate trap handlers for traps taken at trap levels greater than zero (i.e., we are already handling a trap) because not all facilities are available when a trap is taken within a trap.

For example, if a trap handler at trap level 0 takes a memory-related trap (such as a translation miss), the trap handler can assume a higher-level trap handler will take care of the trap; but a higher-level trap handler cannot always make the same assumption. Each half of the trap table contains 512 trap handler slots, one for each trap type shown in Table 2-1.

Each half of the trap table is further divided into two sections, each of which contains 256 hardware traps in the lower section, followed by 256 software traps in the upper section (for the SPARC `Tcc` software trap instructions). Upon receipt of a trap, the UltraSPARC I & II processor jumps to the instructions located in the trap table at the trap table base address (set in the TBA register) plus the offset of the trap level and trap type. There are 8 instructions (32 bytes) at each slot in the table; hence, the trap handler address is calculated as follows:

TL = 0: trap handler address = TBA + (trap type x 32)

TL > 0: trap handler address = TBA + 512 + (trap type x 32)

As a side note, space is reserved in the trap table so that trap handlers for SPARC that register clean, spill, and fill (register window operations) can actually b

longer than 8 instructions. This allows branchless inline handlers to be implemented such that the entire handler fits within the trap table slot.

Figure 2.3 shows the UltraSPARC I & II trap table layout.

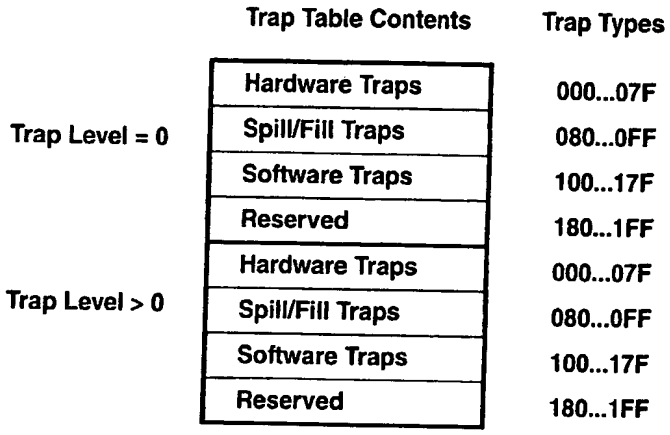

Figure 2.3 UltraSPARC I & II Trap Table Layout

2.2.3.5 Software Traps

Software traps are initiated by the SPARC trap instruction, `Tcc`. The opcode for the trap instruction includes a 6-bit software trap number, which indexes into the software portion of the trap table. Software traps are used primarily for system calls in the Solaris kernel.

There are three software traps for system calls: one for native system calls, one for 32-bit system calls (when 32-bit applications are run on a 64-bit kernel), and one for SunOS 4.x binary compatibility system calls. System calls vector through a common trap by setting the system call number in a global register and then issuing a trap instruction. We discuss regular systems calls in more detail in "System Calls" on page 44.

There are also several ultra-fast system calls implemented as their own trap. These system calls pass their simple arguments back and forth via registers. Because the system calls don't pass arguments on the stack, much less of the process state needs to be saved during transition into kernel mode, resulting in a much faster system call implementation. The fast system calls (e.g., `get_hrestime`) are time-related calls.

Table 2-2 lists UltraSPARC software traps, including ultra-fast system calls.

Table 2-2 UltraSPARC Software Traps

Trap Definition	Trap Type Value	Priority
Trap instruction (SunOS 4.x syscalls)	100	16
Trap instruction (user breakpoints)	101	16
Trap instruction (divide by zero)	102	16
Trap instruction (flush windows)	103	16
Trap instruction (clean windows)	104	16
Trap instruction (do unaligned references)	106	16
Trap instruction (32-bit system call)	108	16
Trap instruction (set trap0)	109	16
Trap instructions (user traps)	110 – 123	16
Trap instructions (get_hrtime)	124	16
Trap instructions (get_hrvtime)	125	16
Trap instructions (self_xcall)	126	16
Trap instructions (get_hrestime)	127	16
Trap instructions (trace)	130-137	16
Trap instructions (64-bit system call)	140	16

2.2.3.6 A Utility for Trap Analysis

An unbundled tool, `trapstat`, dynamically monitors trap activity. The tool monitors counts of each type of trap for each processor in the system during an interval specified as the argument. It is currently implemented on UltraSPARC and Intel x86 processor architectures, on Solaris 7 and later releases.

You can download `trapstat` from the website for this book: `http://www.solarisinternals.com`. Simply untar the archive and install the driver with the `add_drv` command.

Note: `trapstat` *is not supported by Sun. Do not use it on production machines because it dynamically loads code into the kernel.*

```
# tar xvf trapstat28.tar
-r-xr-xr-x   0/2       5268 Jan 31 03:57 2000 /usr/bin/trapstat
-rwxrwxr-x   0/1      33452 Feb 10 23:17 2000 /usr/bin/sparcv7/trapstat
-rwxrwxr-x   0/1      40432 Feb 10 23:16 2000 /usr/bin/sparcv9/trapstat
-rw-rw-r--   0/1      21224 Sep  8 17:28 1999 /usr/kernel/drv/trapstat
-rw-r--r--   0/1        188 Aug 31 10:06 1999 /usr/kernel/drv/trapstat.conf
-rw-rw-r--   0/1      37328 Sep  8 17:28 1999 /usr/kernel/drv/sparcv9/trapstat
# add_drv trapstat
```

Once `trapstat` is installed, use it to analyze the traps taken on each processor installed in the system.

```
# trapstat 3
vct  name                |    cpu0    cpu1
---------------------------+----------------
 24  cleanwin             |    3636    4285
 41  level-1              |      99       1
 45  level-5              |       1       0
 46  level-6              |      60       0
 47  level-7              |      23       0
 4a  level-10             |     100       0
 4d  level-13             |      31      67
 4e  level-14             |     100       0
 60  int-vec              |     161      90
 64  itlb-miss            |    5329   11128
 68  dtlb-miss            |   39130   82077
 6c  dtlb-prot            |       3       2
 84  spill-1-normal       |    1210     992
 8c  spill-3-normal       |     136     286
 98  spill-6-normal       |    5752   20286
 a4  spill-1-other        |     476    1116
 ac  spill-3-other        |    4782    9010
 c4  fill-1-normal        |    1218     752
 cc  fill-3-normal        |    3725    7972
 d8  fill-6-normal        |    5576   20273
103  flush-wins           |      31       0
108  syscall-32           |    2809    3813
124  getts                |    1009    2523
127  gethrtime            |    1004     477
---------------------------+----------------
ttl                       |   76401  165150
```

The example above shows the traps taken on a two-processor UltraSPARC-II-based system. The first column shows the trap type, followed by an ASCII description of the trap type. The remaining columns are the trap counts for each processor.

We can see that most trap activities in the SPARC are register clean, spill, and fill traps—they perform SPARC register window management. The level-1 through level 14 and int-vec rows are the interrupt traps. The itlb-miss, dtlb-miss, and dtlb-prot rows are the UltraSPARC memory management traps, which occur each time a TLB miss or protection fault occurs. (More on UltraSPARC memory management in "The UltraSPARC-I and -II HAT" on page 193.) At the bottom of the output we can see the system call trap for 32-bit systems calls and two special ultra-fast system calls (`getts` and `gethrtime`), which each use their own trap.

The SPARC V9 Architecture Manual [30] provides a full reference for the implementation of UltraSPARC traps. We highly recommend this text for specific implementation details on the SPARC V9 processor architecture.

2.3 Interrupts

An interrupt is the mechanism that a device uses to signal the kernel that it needs attention and some immediate processing is required on behalf of that device. Solaris services interrupts by context-switching out the current thread running on a processor and executing an *interrupt handler* for the interrupting device. For example, when a packet is received on a network interface, the network controller initiates an interrupt to begin processing the packet.

2.3.1 Interrupt Priorities

Solaris assigns priorities to interrupts to allow overlapping interrupts to be handled with the correct precedence; for example, a network interrupt can be configured to have a higher priority than a disk interrupt.

The kernel implements 15 interrupt priority levels: level 1 through level 15, where level 15 is the highest priority level. On each processor, the kernel can mask interrupts below a given priority level by setting the processor's interrupt level. Setting the interrupt level blocks all interrupts at the specified level and lower. That way, when the processor is executing a level 9 interrupt handler, it does not receive interrupts at level 9 or below; it handles only higher-priority interrupts.

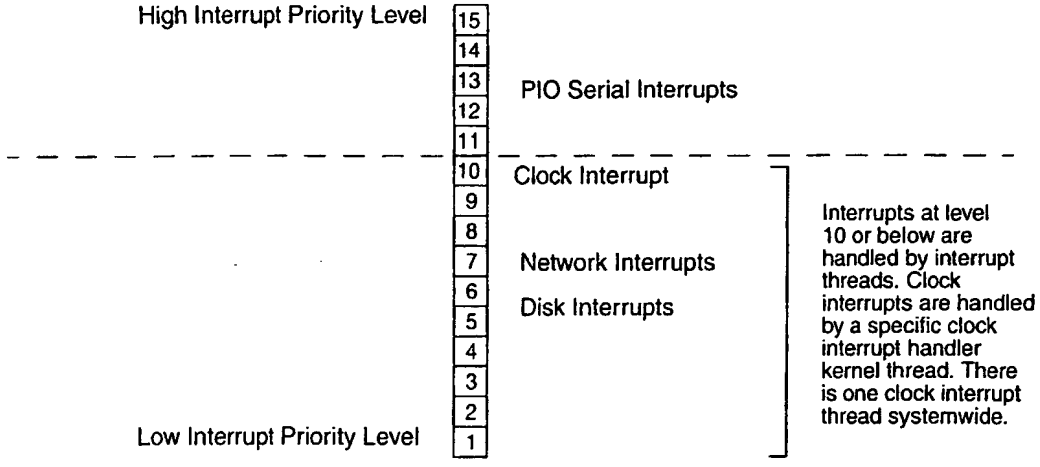

Figure 2.4 Solaris Interrupt Priority Levels

Interrupts that occur with a priority level at or lower than the processor's interrupt level are temporarily ignored. An interrupt will not be acknowledged by a processor until the processor's interrupt level is less than the level of the pending

interrupt. More important interrupts have a higher-priority level to give them a better chance to be serviced than lower-priority interrupts.

Figure 2.4 illustrates interrupt priority levels.

2.3.1.1 Interrupts as Threads

Interrupt priority levels can be used to synchronize access to critical sections used by interrupt handlers. By raising the interrupt level, a handler can ensure exclusive access to data structures for the specific processor that has elevated its priority level. This is in fact what early, uniprocessor implementations of UNIX systems did for synchronization purposes.

But masking out interrupts to ensure exclusive access is expensive; it blocks other interrupt handlers from running for a potentially long time, which could lead to data loss if interrupts are lost because of overrun. (An overrun condition is one in which the volume of interrupts awaiting service exceeds the system's ability to queue the interrupts.) In addition, interrupt handlers using priority levels alone cannot block, since a deadlock could occur if they are waiting on a resource held by a lower-priority interrupt.

For these reasons, the Solaris kernel implements most interrupts as asynchronously created and dispatched high-priority threads. This implementation allows the kernel to overcome the scaling limitations imposed by interrupt blocking for synchronizing data access and thus provides low-latency interrupt response times.

Interrupts at priority 10 and below are handled by Solaris threads. These interrupt handlers can then block if necessary, using regular synchronization primitives such as mutex locks. Interrupts, however, must be efficient, and it is too expensive to create a new thread each time an interrupt is received. For this reason, each processor maintains a pool of partially initialized interrupt threads, one for each of the lower 9 priority levels plus a systemwide thread for the clock interrupt. When an interrupt is taken, the interrupt uses the interrupt thread's stack, and only if it blocks on a synchronization object is the thread completely initialized. This approach, exemplified in Figure 2.5, allows simple, fast allocation of threads at the time of interrupt dispatch.

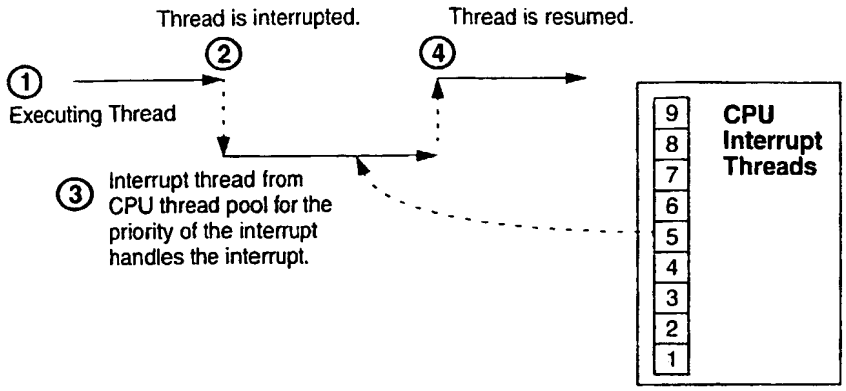

Figure 2.5 Handling Interrupts with Threads

Figure 2.5 depicts a typical scenario when an interrupt with priority 9 or less occurs (level 10 clock interrupts are handled slightly differently). When an interrupt occurs, the interrupt level is raised to the level of the interrupt to block subsequent interrupts at this level (and lower levels). The currently executing thread is interrupted and *pinned* to the processor. A thread for the priority level of the interrupt is taken from the pool of interrupt threads for the processor and is context-switched in to handle the interrupt.

The term *pinned* refers to a mechanism employed by the kernel that avoids context switching out the interrupted thread. The executing thread is pinned under the interrupt thread. The interrupt thread "borrows" the LWP from the executing thread. While the interrupt handler is running, the interrupted thread is pinned to avoid the overhead of having to completely save its context; it cannot run on any processor until the interrupt handler completes or blocks on a synchronization object. Once the handler is complete, the original thread is unpinned and rescheduled.

If the interrupt handler thread blocks on a synchronization object (e.g., a mutex or condition variable) while handling the interrupt, it is converted into a complete kernel thread capable of being scheduled. Control is passed back to the interrupted thread, and the interrupt thread remains blocked on the synchronization object. When the synchronization object is unblocked, the thread becomes runnable and may preempt lower-priority threads to be rescheduled.

The processor interrupt level remains at the level of the interrupt, blocking lower-priority interrupts, even while the interrupt handler thread is blocked. This prevents lower-priority interrupt threads from interrupting the processing of higher-level interrupts. While interrupt threads are blocked, they are pinned to the processor they initiated on, guaranteeing that each processor will always have an interrupt thread available for incoming interrupts.

Level 10 clock interrupts are handled in a similar way, but since there is only one source of clock interrupt, there is a single, systemwide clock thread. Clock interrupts are discussed further in "The System Clock" on page 54.

2.3.1.2 Interrupt Thread Priorities

Interrupts that are scheduled as threads share global dispatcher priorities with other threads. See Chapter 9, "The Solaris Kernel Dispatcher" for a full description of the Solaris dispatcher. Interrupt threads use the top ten global dispatcher priorities, 160 to 169. Figure 2.6 shows the relationship of the interrupt dispatcher priorities with the real-time, system (kernel) threads and the timeshare and interactive class threads.

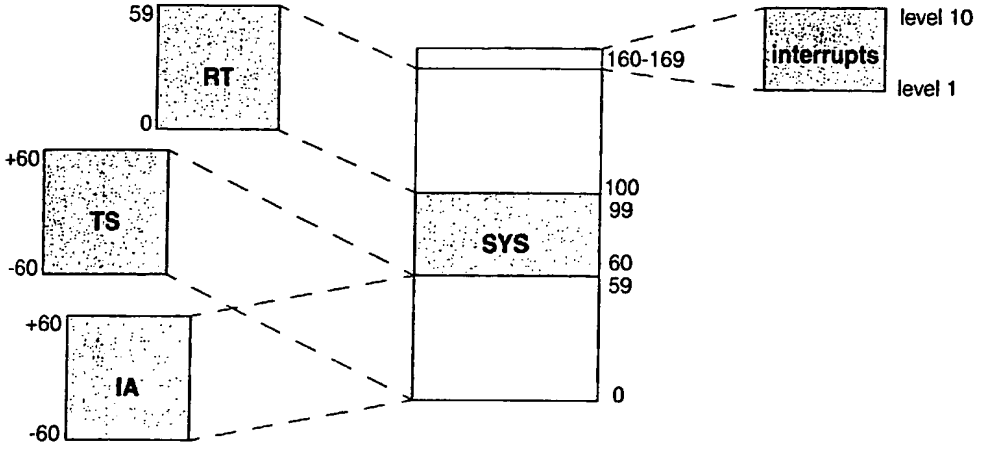

Figure 2.6 Interrupt Thread Global Priorities

2.3.1.3 High-Priority Interrupts

Interrupts above priority 10 block out all lower-priority interrupts until they complete. For this reason, high-priority interrupts need to have an extremely short code path to prevent them from affecting the latency of other interrupt handlers and the performance and scalability of the system. High-priority interrupt threads also cannot block; they can use only the spin variety of synchronization objects. This is due to the priority level the dispatcher uses for synchronization. The dispatcher runs at level 10, thus code running at higher interrupt levels cannot enter the dispatcher. High-priority threads typically service the minimal requirements of the hardware device (the source of the interrupt), then post down a lower-priority software interrupt to complete the required processing.

2.3.1.4 UltraSPARC Interrupts

On UltraSPARC systems (sun4u), the `intr_vector[]` array is a single, system-wide interrupt table for all hardware and software interrupts, as shown in Figure 2.7.

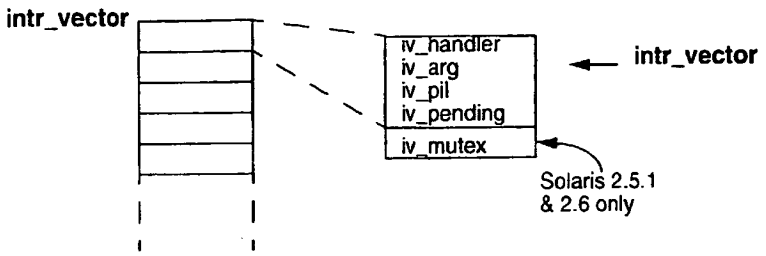

Figure 2.7 Interrupt Table on sun4u Architectures

Interrupts are added to the array through an `add_ivintr()` function. (Other platforms have a similar function for registering interrupts.) Each interrupt registered with the kernel has a unique interrupt number that locates the handler information in the interrupt table when the interrupt is delivered. The interrupt number is passed as an argument to `add_ivintr()`, along with a function pointer (the interrupt handler, `iv_handler`), an argument list for the handler (`iv_arg`), and the priority level of the interrupt (`iv_pil`).

Solaris 2.5.1 and Solaris 2.6 allow for unsafe device drivers—drivers that have not been made multiprocessor safe through the use of locking primitives. For unsafe drivers, a mutex lock locks the interrupt entry to prevent multiple threads from entering the driver's interrupt handler.

Solaris 7 requires that all drivers be minimally MP safe, dropping the requirement for a lock on the interrupt table entry. The `iv_pending` field is used as part of the queueing process; generated interrupts are placed on a per-processor list of interrupts waiting to be processed. The `pending` field is set until a processor prepares to field the interrupt, at which point the `pending` field is cleared.

A kernel `add_softintr()` function adds software-generated interrupts to the table. The process is the same for both functions: use the interrupt number passed as an argument as an index to the `intr_vector[]` array, and add the entry. The size of the array is large enough that running out of array slots is unlikely.

2.3.2 Interrupt Monitoring

You can use the `mpstat`(1M) and `vmstat`(1M) commands to monitor interrupt activity on a Solaris system. `mpstat`(1M) provides interrupts-per-second for each

CPU in the intr column, and interrupts handled on an interrupt thread (low-level interrupts) in the ithr column.

```
# mpstat 3
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    5   0    7   39   12  250   17    9   18    0   725    4   2   0  94
  1    4   0   10  278   83  275   40    9   40    0   941    4   2   0  93
```

2.3.3 Interprocessor Interrupts and Cross-Calls

The kernel can send an interrupt or trap to another processor when it requires another processor to do some immediate work on its behalf. Interprocessor interrupts are delivered through the `poke_cpu()` function; they are used for the following purposes:

- **Preempting the dispatcher** — A thread may need to signal a thread running on another processor to enter kernel mode when a preemption is required (initiated by a clock or timer event) or when a synchronization object is released. Chapter 9, "The Dispatcher," further discusses preemption.
- **Delivering a signal** — The delivery of a signal may require interrupting a thread on another processor.
- **Starting/stopping /proc threads** — The `/proc` infrastructure uses interprocessor interrupts to start and stop threads on different processors.

Using a similar mechanism, the kernel can also instruct a processor to execute a specific low-level function by issuing a processor-to-processor *cross-call*. Cross-calls are typically part of the processor-dependent implementation. UltraSPARC kernels use cross-calls for two purposes:

- **Implementing interprocessor interrupts** — As discussed above.
- **Maintaining virtual memory translation consistency** — Implementing cache consistency on SMP platforms requires the translation entries to be removed from the MMU of each CPU that a thread has run on when a virtual address is unmapped. On UltraSPARC, user processes issuing an unmap operation make a cross-call to each CPU on which the thread has run, to remove the TLB entries from each processor's MMU. Address space unmap operations within the kernel address space make a cross-call to *all* processors for each unmap operation.

Both cross-calls and interprocessor interrupts are reported by `mpstat(1M)` in the xcal column as cross-calls per second.

```
# mpstat 3
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    6   607  246 1100  174   82   84    0  2907   28   5   0  66
  1    0   0    2   218    0 1037  212   83   80    0  3438   33   4   0  62
```

High numbers of reported cross-calls can result from either of the activities mentioned in the preceding section—most commonly, from kernel address space unmap activity caused by file system activity.

2.4 System Calls

Recall from "Access to Kernel Services" on page 27, system calls are interfaces callable by user programs in order to have the kernel perform a specific function (e.g., opening a file) on behalf of the calling thread. System calls are part of the application programming interfaces (APIs) that ship with the operating system; they are documented in Section 2 of the manual pages. The invocation of a system call causes the processor to change from user mode to kernel mode. This change is accomplished on SPARC systems by means of the trap mechanism previously discussed.

2.4.1 Regular System Calls

System calls are referenced in the system through the kernel `sysent` table, which contains an entry for every system call supported on the system. The `sysent` table is an array populated with `sysent` structures, each structure representing one system call, as illustrated in Figure 2.8.

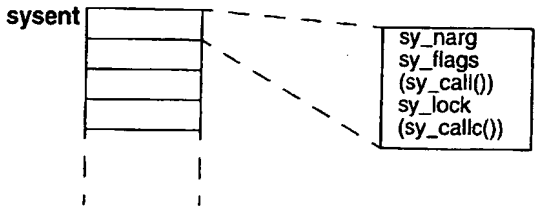

Figure 2.8 The Kernel System Call Entry (`sysent`) Table

The array is indexed by the system call number, which is established in the `/etc/name_to_sysnum` file. Using an editable system file provides for adding system calls to Solaris without requiring kernel source and a complete kernel build. Many system calls are implemented as dynamically loadable modules that are loaded into the system when the system call is invoked for the first time. Loadable system calls are stored in the `/kernel/sys` and `/usr/kernel/sys` directories.

The system call entry in the table provides the number of arguments the system call takes (`sy_narg`), a flag field (`sy_flag`), and a reader/writer lock

(sy_lock) for loadable system calls. The system call itself is referenced through a function pointer: sy_call or sy_callc.

Historical Aside: The fact that there are two entries for the system call functions is the result of a rewriting of the system call argument-passing implementation, an effort that first appeared in Solaris 2.4. Earlier Solaris versions passed system call arguments in the traditional UNIX way: bundling the arguments into a structure and passing the structure pointer (*uap* is the historical name in UNIX implementations and texts; it refers to a *user argument pointer*). Most of the system calls in Solaris have been rewritten to use the C language argument-passing convention implemented for function calls. Using that convention provided better overall system call performance because the code can take advantage of the argument-passing features inherent in the register window implementation of SPARC processors (using the *in* registers for argument passing—refer to [31] for a description of SPARC register windows).

sy_call represents an entry for system calls and uses the older uap pointer convention, maintained here for binary compatibility with older versions of Solaris. sy_callc is the function pointer for the more recent argument-passing implementation. The newer C style argument passing has shown significant overall performance improvements in system call execution—on the order of 30 percent in some cases.

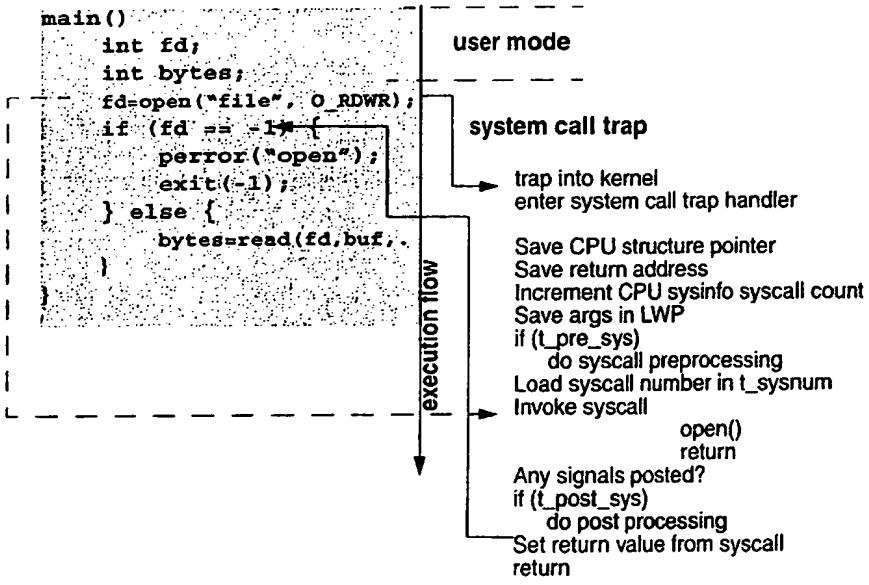

Figure 2.9 System Call Execution

The execution of a system call results in the software issuing a trap instruction, which is how the kernel is entered to process the system call. The trap handler for the system call is entered, any necessary preprocessing is done, and the system call is executed on behalf of the calling thread. The flow is illustrated in Figure 2.9.

When the trap handler is entered, the trap code saves a pointer to the CPU structure of the CPU on which the system call will execute, saves the return address, and increments a system call counter maintained on a per-CPU basis. The number of system calls per second is reported by `mpstat(1M)` (*syscl* column) for per-CPU data; systemwide, the number is reported by `vmstat(1M)` (*sy* column).

Two flags in the kernel thread structure indicate that pre-system call or post-system call processing is required. The `t_pre_sys` flag (preprocessing) is set for things like `truss(1)` command support (system call execution is being traced) or microstate accounting being enabled. Post-system-call work (`t_post_sys`) may be the result of `/proc` process tracing, profiling enabled for the process, a pending signal, or an exigent preemption. In the interim between pre- and postprocessing, the system call itself is executed.

2.4.2 Fast Trap System Calls

The overhead of the system call framework is nontrivial; that is, there is some inherent latency with all system calls because of the system call setup process we just discussed. In some cases, we want to be able to have fast, low-latency access to information, such as high-resolution time, that can only be obtained in kernel mode. The Solaris kernel provides a fast system call framework so that user processes can jump into protected kernel mode to do minimal processing and then return, without the overhead of the full system call framework. This framework can only be used when the processing required in the kernel does not significantly interfere with registers and stacks. Hence, the fast system call does not need to save all the state that a regular system call does before it executes the required functions.

Only a few fast system calls are implemented in Solaris versions up to Solaris 7: `gethrtime()`, `gethrvtime()`, and `gettimeofday()`. These functions return time of day and processor CPU time. They simply trap into the kernel to read a single hardware register or memory location and then return to user mode.

Table 2-3 compares the average latency for the `getpid()/time()` system calls and two fast system calls. For reference, the latency of a standard function call is also shown. Times were measured on a 300 MHz Ultra2. Note that the latency of the fast system calls is about five times lower than that of an equivalent regular system call.